On sessions and infinite data

On Sessions and Infinite Data ⋆

Paula Severi[1], Luca Padovani[2], Emilio Tuosto[1], and Mariangiola Dezani-Ciancaglini[2]

[1] Department of Computer Science, University of Leicester, UK
[2] Dipartimento di Informatica, Università di Torino, Italy

Abstract. We investigate some subtle issues that arise when programming distributed computations over infinite data structures. To do this, we formalise a calculus that combines a call-by-name functional core with session-based communication primitives and that allows session operations to be performed "on demand". We develop a typing discipline that guarantees both normalisation of expressions and progress of processes and that uncovers an unexpected interplay between evaluation and communication.

1 Introduction

Infinite computations have long lost their negative connotation. Two paradigmatic contexts in which they appear naturally are reactive systems [17] and lazy functional programming. The former contemplate the use of infinite computations in order to capture *non-transformational* computations, that is computations that cannot be expressed in terms of transformations from inputs to outputs; rather, computations of reactive systems are naturally modelled in terms of ongoing interactions with the environment. Lazy functional programming is acknowledged as a paradigm that fosters software modularity [13] and enables programmers to specify computations over possibly infinite data structures in elegant and concise ways. Nowadays, the synergy between these two contexts has a wide range of potential applications, including stream-processing networks, real-time sensor monitoring, and internet-based media services.

Nonetheless, not all diverging programs – those engaged in an infinite sequence of possibly intertwined computations and communications – are necessarily useful. There exist degenerate forms of divergence where programs do not produce results, in terms of observable data or performed communications. In this paper we investigate the issue by proposing a calculus for expressing computations over possibly infinite data types and involving message passing. The calculus – called SID after Sessions with Infinite Data – combines a call-by-name functional core (inspired by Haskell) with multi-threading and session-based communication primitives.

In the remainder of this section we provide an informal introduction to SID and its key features by means of a few examples. The formal definition of the calculus, of the type system, and its properties are given in the remaining sections. A simple instance of computation producing an infinite data structure is given by

$$\texttt{from}\, x = \langle x, \texttt{from}\, (x+1)\rangle$$

⋆ Paula Severi has been supported by a Daphne Jackson fellowship sponsored by EPSRC and her department. All authors have been supported by the ICT COST European project called *Behavioural Types for Reliable Large-Scale Software Systems* (BETTY, COST Action IC1201).

where the function `from` applied to a number n produces the stream (infinite list) $\langle n, \langle n+1, \langle n+2, \cdots \rangle \rangle \rangle$ of integers starting from n. We can think of this list as abstracting the frames of a video stream, or the samples taken from a sensor.

The key issue we want to address is how infinite data can be exchanged between communicating threads. The most straightforward way of doing this in SID is to take advantage of lazy evaluation. For instance, the SID process

$$x \Leftarrow \big(\texttt{send}\ c^+\ (\texttt{from}\ 0)\big) \texttt{>>=}\ f \quad \mid \quad y \Leftarrow \texttt{recv}\ c^- \texttt{>>=}\ g$$

represents two threads x and y running in parallel and connected by a session c, of which thread x owns one endpoint c^+ and thread y the corresponding peer c^-. Thread x sends a stream of natural numbers on c^+ and continues as $f\ c^+$, where f is left unspecified. Thread y receives the stream from c^- and continues as $(g\ \langle \texttt{from}\ 0, c^- \rangle)$. The *bind* operator `_ >>= _` models sequential composition and has the exact same semantics as in Haskell. In particular, it applies the rhs to the result of the action on its lhs. The result of sending a message on the endpoint a^+ is the endpoint itself, while the result of receiving a message from the endpoint a^- is a pair consisting of the message and the endpoint. In this example, the *whole stream* is sent *at once* in a single interaction between x and y. This behaviour is made possible by the fact that SID evaluates expressions *lazily*: the message $(\texttt{from}\ 0)$ is not evaluated until it is used by the receiver.

In principle, exchanging "infinite" messages such as $(\texttt{from}\ 0)$ between different threads is no big deal. In the real world, though, this interaction poses non-trivial challenges: the message consists in fact of a mixture of data (the parts of the messages that have already been evaluated, like the constant 0) and code (which lazily computes the remaining parts when necessary, like `from`). This observation suggests an alternative, more viable modelling of this interaction whereby the sender unpacks the stream element-wise, sends each element of the stream as a separate message, and the receiver gradually reconstructs the stream as each element arrives at destination. This modelling is intuitively simpler to realise (especially in a distributed setting) because the messages exchanged at each communication are ground values rather than a mixture of data and code. In SID we can model this as a process

$$\mathit{prod} \Leftarrow \texttt{stream}\ c^+\ (\texttt{from}\ 0) \quad \mid \quad \mathit{cons} \Leftarrow \texttt{display}_0\ c^-$$

where the functions `stream` and $\texttt{display}_0$ are defined as:

$$\begin{aligned} \texttt{stream}\ y\ \langle x, xs \rangle &= \texttt{send}\ y\ x \texttt{>>=}\ \lambda y'.\texttt{stream}\ y'\ xs \\ \texttt{display}_0\ y \qquad &= \texttt{recv}\ y \texttt{>>=}\ \lambda \langle z, y' \rangle.\texttt{display}_0\ y' \texttt{>>=}\ \lambda zs.g\ \langle z, zs \rangle \end{aligned} \tag{1.1}$$

The syntax $\lambda\langle \_, \_ \rangle.e$ is just syntactic sugar for a function that performs pattern matching on the argument, which must be a pair, in order to access its components. In `stream`, pattern matching is used for accessing and sending each element of the stream separately. In $\texttt{display}_0$, the pair $\langle z, y' \rangle$ contains the received head z of the stream along with the continuation y' of the session endpoint from which the element has been received. The recursive call $\texttt{display}_0\ y'$ retrieves the tail of the stream zs, which is then combined with the head z and passed as an argument to g.

The code of $\texttt{display}_0$ looks reasonable at first, but conceals a subtle and catastrophic pitfall: the recursive call $\texttt{display}_0\ y'$ is in charge of receiving the *whole* tail zs,

which is an infinite stream itself, and therefore it involves an infinite number of synchronisations with the producing thread! This means that $\texttt{display}_0$ will hopelessly diverge striving to receive the whole stream before releasing control to g. This is a known problem which has led to the development of primitives (such as `unsafeInterleaveIO` in Haskell or `delayIO` in [23]) that allow the execution of I/O actions to interleave with their continuation. In this paper, we call such primitive `future`, since its semantics is also akin to that of *future variables* [25]. Intuitively, an expression $\texttt{future}\ e \gg\!= \lambda x.(g\ x)$ allows g to reduce even if e, which typically involves I/O, has not been completely performed. The variable x acts as a placeholder for the result of e; if g needs to inspect the structure of x, its evaluation is suspended until e produces enough data. Using `future` we can amend the definition of $\texttt{display}_0$ thus

$$\texttt{display}\ y = \texttt{recv}\ y \gg\!= \lambda\langle z, y'\rangle.\texttt{future}\ (\texttt{display}\ y') \gg\!= \lambda zs.g\ \langle z, zs\rangle \qquad (1.2)$$

and obtain one that allows g to start processing the stream as its elements come through the connection with the producer thread. The type system that we develop in this paper allows us to reason on sessions involving the exchange of infinite data and when such exchanges can be done "productively". In particular, our type system flags $\texttt{display}_0$ in (1.1) as ill typed, while it accepts `display` in (1.2) as well typed. To do so, the type system uses a modal operator $\bullet$ related to the normalisability of expressions. As hinted by the examples (1.1) and (1.2), this operator plays a major role in the type of `future`.

Related Work. To the best of our knowledge, SID is the first calculus that combines *session-based* communication primitives [12,29] with a *call-by-need* operational semantics [30,1,18] guaranteeing progress of processes exchanging infinite data. The operational semantics of related session calculi that appear in the literature is call-by-value, e.g. [11,9,28] making them unsuitable for handling potentially infinite data, such as streams. In the context of communication-centric calculi, SSCC [7] offers an explicit primitive to deal with streams. Our language enables the modelling of more intricate interactions between infinite data structures and infinite communications. Besides, the type system of SSCC considers only finite sessions types and does not guarantee progress of processes.

Following [19], we use a modal operator $\bullet$ to restrict the application of the fixed point operator and exclude degenerate forms of divergence. This paper is an improvement over past typed lambda calculi with a temporal modal operator in two respects. Firstly, we do not need any subtyping relation as in [19] and secondly SID programs are not cluttered with constructs for the introduction and elimination of individuals of type $\bullet$ as in [14,26,3,15,14]. A weak criterion to ensure productivity of infinite data is the *guardedness condition* [5]. We do not need such condition because we can type more normalising expressions (such as `display` in (1.2)) using the modal operator $\bullet$.

Futures originated in functional programming and related paradigms for parallelising a program [10]. The call-by-need λ-calculus with futures in [25] is used for studying contextual equivalence and has no type system.

In the session calculi literature, the word "progress" has two different meanings. Sometimes it is synonym of deadlock freedom [2], at other times it means lock freedom, i.e. that each offered communication in an open session eventually happens [8,20,4].

Table 1. Syntax of expressions and processes.

		Expression			**Process**
e	::=		P	::=	
	`k`	(constant)		`0`	(idle process)
	u	(name)		$x \Leftarrow e$	(thread)
	$\lambda x.e$	(abstraction)		`server` $a\ e$	(server)
	ee	(application)		$P \mid P$	(parallel)
	`split` e `as` x,y `in` e	(pair splitting)		$(\nu X)P$	(restriction)

`k` ::= `unit` | `return` | `open` | `send` | `recv` | `future` | `pair` | `bind`

Typed SID processes cannot be stuck, and if they do not terminate they communicate and/or generate new threads infinitely often. This means that the property of progress satisfied by our calculus is stronger than that of [2] and weaker than that of [8,20,4].

Contributions and Outline. The SID calculus, defined in Section 2, combines in an original way standard constructs from the λ-calculus and process algebras in the spirit of [12,11]. The type system, given in Section 3, has the novelty of using the modal operator $\bullet$ to control the recursion of programs that perform communications. To the best of our knowledge, the interplay between $\bullet$ and the type of `future` is investigated here for the first time. The properties of our framework, presented in Section 4, include subject reduction (Theorem 1), normalisation of expressions (Theorem 2), progress and confluence of processes (Theorems 4, 5). Additional examples, definitions, and proofs can be found in the technical report [27].

2 The SID Calculus

We use an infinite set of *channels* a, b, c and a disjoint, infinite set of *variables* x, y. We distinguish between two kinds of channels: *shared channels* are public service identifiers that can only be used to initiate sessions; *session channels* represent private sessions on which the actual communications take place. We distinguish the two *endpoints* of a session channel c by means of a *polarity* $p \in \{+,-\}$ and write them as c^+ and c^-. We write $\overline{p}$ for the dual polarity of p, where $\overline{+} = -$ and $\overline{-} = +$, and we say that c^p is the *peer endpoint* of $c^{\overline{p}}$. A *bindable name* X is either a channel or a variable and a *name* u is either a bindable name or an endpoint.

The syntax of *expressions* and *processes* is given in Table 1. In addition to the usual constructs of the λ-calculus, expressions include constants, ranged over by `k`, and pair splitting. Constant are the unitary value `unit`, the pair constructor `pair`, the primitives for session initiation and communication `open`, `send`, and `recv` [12,11], the monadic operations `return` and `bind` [23], and a primitive `future` to defer computations [22,21]. We do not need a primitive constant for the fixed point operator because it can be expressed and typed inside the language. For simplicity, we do not include primitives for branching and selection typically found in session calculi. They are straightforward to add and do not invalidate any of the results. Expressions are subject to the usual conventions of the λ-calculus. In particular, we assume that the bodies

Table 2. Reduction semantics of expressions and processes.

Reduction of expressions

$$[\text{R-BETA}]\quad (\lambda x.e)\ f \longrightarrow e\{f/x\} \qquad [\text{R-BIND}]\quad \texttt{return}\ f\ \texttt{>>=}\ e \longrightarrow e\,f$$

$$[\text{R-SPLIT}]\quad \texttt{split}\ \langle e_1\,,e_2\rangle\ \texttt{as}\ x,y\ \texttt{in}\ e \longrightarrow e\{e_1,e_2/x,y\} \qquad [\text{R-CTXT}]\quad \frac{e \longrightarrow f}{\mathscr{E}[e] \longrightarrow \mathscr{E}[f]}$$

Reduction of processes

$$[\text{R-OPEN}]\quad \texttt{server}\ a\ e \mid x \Leftarrow \mathscr{C}[\texttt{open}\ a] \longrightarrow \texttt{server}\ a\ e \mid (\nu cy)(x \Leftarrow \mathscr{C}[\texttt{return}\ c^+] \mid y \Leftarrow e\ c^-)$$

$$[\text{R-COMM}]\quad x \Leftarrow \mathscr{C}[\texttt{send}\ a^p\ e] \mid y \Leftarrow \mathscr{C}'[\texttt{recv}\ a^{\overline{p}}] \longrightarrow x \Leftarrow \mathscr{C}[\texttt{return}\ a^p] \mid y \Leftarrow \mathscr{C}'[\texttt{return}\ \langle e\,,a^{\overline{p}}\rangle]$$

$$[\text{R-FUTURE}]\quad x \Leftarrow \mathscr{C}[\texttt{future}\ e] \longrightarrow (\nu y)(x \Leftarrow \mathscr{C}[\texttt{return}\ y] \mid y \Leftarrow e)$$

$$[\text{R-RETURN}]\quad (\nu x)(x \Leftarrow \texttt{return}\ e \mid P) \longrightarrow P\{e/x\}$$

$$[\text{R-THREAD}]\quad \frac{e \longrightarrow f}{x \Leftarrow e \longrightarrow x \Leftarrow f} \qquad [\text{R-NEW}]\quad \frac{P \longrightarrow Q}{(\nu X)P \longrightarrow (\nu X)Q} \qquad [\text{R-PAR}]\quad \frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} \qquad [\text{R-CONG}]\quad \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

of abstractions extend as much as possible to the right, that applications associate to the left, and we use parentheses to disambiguate the notation when necessary. Following established notation, we write $\langle e\,,f\rangle$ in place of $\texttt{pair}\ e\ f$, $\lambda\langle x_1\,,x_2\rangle.e$ in place of $\lambda x.\texttt{split}\ x\ \texttt{as}\ x_1,x_2\ \texttt{in}\ e$, and $e\ \texttt{>>=}\ f$ in place of $\texttt{bind}\ e\ f$.

A process can be either the idle process $\texttt{0}$ that performs no action, a thread $x \Leftarrow e$ with name x and body e that evaluates the body and binds the result to x in the rest of the system, a $\texttt{server}\ a\ e$ that waits for session initiations on the shared channel a and spawns a new thread computing e at each connection, the parallel composition of processes, and the restriction of a bindable name. In processes, restrictions bind tighter than parallel composition and we may abbreviate $(\nu X_1)\cdots(\nu X_n)P$ with $(\nu X_1\cdots X_n)P$.

We have that $\texttt{split}\ f\ \texttt{as}\ x,y\ \texttt{in}\ e$ binds both x and y in e and $(\nu a)P$ binds a^+ and a^- within P in addition to a. The definitions of *free* and *bound* names follow as expected. We identify expressions and processes up to renaming of bound names.

The operational semantics of expressions is defined in the upper half of Table 2. Expressions reduce according to a standard *call-by-name* semantics, for which we define the *evaluation contexts for expressions* below:

$$\mathscr{E} ::= [\,] \mid \mathscr{E}\,e \mid \texttt{split}\ \mathscr{E}\ \texttt{as}\ x,y\ \texttt{in}\ e \mid \texttt{open}\ \mathscr{E} \mid \texttt{send}\ \mathscr{E} \mid \texttt{recv}\ \mathscr{E} \mid \texttt{bind}\ \mathscr{E}$$

Note that evaluation contexts do not allow to reduce pair components or an expression e in `bind` f e, `return` e, `future` e, `send` a^p e. We say that e is in *normal form* if there is no f such that $e \longrightarrow f$.

The operational semantics of processes is given by a structural congruence relation $\equiv$, which we leave undetailed since it is essentially the same as that of the π-calculus, and a reduction relation, defined in the bottom half of Table 2. The *evaluation contexts for processes* are defined as

$$\mathscr{C} ::= [\,] \mid \mathscr{C} \gg\!= e$$

and force the left-to-right execution of monadic actions, as usual.

Rules [R-OPEN] and [R-COMM] model session initiation and communication, respectively. According to [R-OPEN], a client thread opens a connection with a server a. In the reduct, a fresh session channel c is created, the `open` in the client reduces to the return of c^+ and a copy of the server is spawned into a new thread that has a fresh name y and a body obtained from that of the server applied to c^-. According to [R-COMM], two threads communicate if one is ready to send some message e on a session endpoint a^p and the other is waiting for a message from the peer endpoint $a^{\overline{p}}$. As in [11], the communication primitives return the session endpoint being used, with the difference that in our case the results are monadic actions. In particular, the result for the sender is the same session endpoint and the result for the receiver is a pair consisting of the received message and the session endpoint.

Rules [R-FUTURE] and [R-RETURN] deal with futures. The former spawns an I/O action e in a separate thread y, so that the spawner is able to reduce (using [R-BIND]) even if e has not been executed yet. The name y of the spawned thread can be used as a placeholder for the value yielded by e. Rule [R-RETURN] deals with a future variable x that has been evaluated to `return` e. In this case, x is replaced by e everywhere within its scope.

Rule [R-THREAD] lifts reduction of expressions to reduction of threads. The remaining rules close reduction under restrictions, parallel compositions, and structural congruence, as expected.

3 Typing SID

We now develop a typing discipline for SID. The challenge comes from the fact that the calculus allows a mixture of pure computations (handling data) and impure computations (doing I/O). In particular, SID programs can manipulate potentially infinite data while performing I/O operations that produce/consume pieces of such data as shown by the examples of Section 1. Some ingredients of the type system are easily identified from the syntax of the calculus. We have a core type language with unit, products, and arrows. As in [11], we distinguish between *unlimited* and *linear* arrows for there sometimes is the need to specify that certain functions must be applied exactly once. As in Haskell [23,21], we use the `IO` type constructor to denote monadic I/O actions. For shared and session channels we respectively introduce channel types and session types [12]. Finally, following [19], we introduce the *delay* type constructor $\bullet$, so that an expression of type $\bullet t$ denotes a value of type t that is available "at the next moment in time". This constructor is key to control recursion and attain normalisation of expres-

Table 3. Syntax of Pseudo-types and Pseudo-session types.

$t ::=^{coind}$		**Pseudo-type**	$T ::=^{coind}$		**Pseudo-session type**
	$\mathtt{B}$	(basic type)		$\mathtt{end}$	(end)
$\mid$	T	(session type)	$\mid$	$?t.T$	(input)
$\mid$	$\langle T \rangle$	(shared channel type)	$\mid$	$!t.T$	(output)
$\mid$	$t \times t$	(product)	$\mid$	$\bullet T$	(delay)
$\mid$	$t \to t$	(arrow)			
$\mid$	$t \multimap t$	(linear arrow)			
$\mid$	$\mathtt{IO}\, t$	(input/output)			
$\mid$	$\bullet t$	(delay)			

sions. Moreover, the type constructors $\bullet$ and $\mathtt{IO}$ interact in non-trivial ways as shown later by the type of `future`.

3.1 Types

The syntax of *pseudo-types* and *pseudo-session types* is given by the grammar in Table 3, whose productions are meant to be interpreted coinductively. A pseudo (session) type is a possibly infinite tree, where each internal node is labelled by a type constructor and has as many children as the arity of the constructor. The leaves of the tree (if any) are labelled by either basic types or $\mathtt{end}$. We use a coinductive syntax to describe infinite data structures (such as streams) and arbitrarily long protocols, such as the one betwen sender and receiver in Section 1.

We distinguish between unlimited pseudo-types (those denoting expressions that can be used any number of times) from linear pseudo-types (those denoting expressions that must be used exactly once). Let lin be the smallest predicate defined by

$$\mathsf{lin}(?t.T) \quad \mathsf{lin}(!t.T) \quad \mathsf{lin}(t \multimap s) \quad \mathsf{lin}(\mathtt{IO}\, t) \quad \frac{\mathsf{lin}(t)}{\mathsf{lin}(t \times s)} \quad \frac{\mathsf{lin}(s)}{\mathsf{lin}(t \times s)} \quad \frac{\mathsf{lin}(t)}{\mathsf{lin}(\bullet t)}$$

The word "smallest" in the above definition is crucial. For example lin does not hold for the type $\bullet^\infty$, because $\bullet^\infty$ does not belong to the smallest set satisfying the above clauses. We say that t is *linear* if $\mathsf{lin}(t)$ holds and that t is *unlimited*, written $\mathsf{un}(t)$, otherwise. Note that all I/O actions are linear, since they may involve communications on session channels which are linear resources.

Definition 1 (Types). *A pseudo (session) type t is a* (session) type *if:*

1. *For each sub-term $t_1 \to t_2$ of t such that $\mathsf{un}(t_2)$ we have $\mathsf{un}(t_1)$.*
2. *For each sub-term $t_1 \multimap t_2$ of t we have $\mathsf{lin}(t_2)$.*
3. *The tree representation of t is regular, namely it has finitely many distinct sub-trees.*
4. *Every infinite path in the tree representation of t has infinitely many $\bullet$'s.*

All conditions except possibly 4 are natural. Condition 1 essentially says that unlimited functions are *pure*, namely they do not have side effects. Indeed, an unlimited function (one that does not contain linear names) that accepts a linear argument

should return a linear result. Condition 2 states that a linear function (one that may contain linear names) always yields a linear result. This is necessary to keep track of the presence of linear names in the function, even when the function is applied and its linear arrow type eliminated. For example, consider z of type $\mathtt{Nat} \multimap \mathtt{Nat}$ and both y and w of type $\mathtt{Nat}$, then without Condition 2 we could type $(\lambda x.y)(z\ w)$ with $\mathtt{Nat}$. This would be incorrect, because it discharges the expression $(z\ w)$ involving the linear name z. Condition 3 implies that we only consider types admitting a finite representation, for example using the well-known "μ notation" for expressing recursive types (for the relation between regular trees and recursive types we refer to [24, Ch. 20]). We define infinite types as trees satisfying a given recursive equation, for which the existence and uniqueness of a solution follow from known results [6]. For example, there are unique pseudo-types $\mathtt{S}'_{\mathtt{Nat}}$, $\mathtt{S}_{\mathtt{Nat}}$, and $\bullet^\infty$ that respectively satisfy the equations $\mathtt{S}'_{\mathtt{Nat}} = \mathtt{Nat} \times \mathtt{S}'_{\mathtt{Nat}}$, $\mathtt{S}_{\mathtt{Nat}} = \mathtt{Nat} \times \bullet\mathtt{S}_{\mathtt{Nat}}$, and $\bullet^\infty = \bullet\bullet^\infty$. *En passant*, note that linearity is decidable on types due to Condition 3.

Condition 4 intuitively means that not all parts of an infinite data structure can be available at once: those whose type is prefixed by a $\bullet$ are necessarily "delayed" in the sense that recursive calls on them must be deeper. For example, $\mathtt{S}_{\mathtt{Nat}}$ is a type that denotes streams of natural numbers where each subsequent element of the stream is delayed by one $\bullet$ compared to its predecessor. Instead $\mathtt{S}'_{\mathtt{Nat}}$ is not a type: it would denote an infinite stream of natural numbers, whose elements are all available right away. Similarly, $\mathtt{Out}_{\mathtt{Nat}}$ and $\mathtt{In}_{\mathtt{Nat}}$ defined by $\mathtt{Out}_{\mathtt{Nat}} = \mathtt{!Nat}.\bullet\mathtt{Out}_{\mathtt{Nat}}$ and $\mathtt{In}_{\mathtt{Nat}} = \mathtt{?Nat}.\bullet\mathtt{In}_{\mathtt{Nat}}$ are session types, while $\mathtt{O}'_{\mathtt{Nat}}$ and $\mathtt{I}'_{\mathtt{Nat}}$ defined by $\mathtt{O}'_{\mathtt{Nat}} = \mathtt{!Nat.Out}_{\mathtt{Nat}}$ and $\mathtt{I}'_{\mathtt{Nat}} = \mathtt{?Nat.I}'_{\mathtt{Nat}}$ are not. The type $\bullet^\infty$ is somehow degenerate in that it contains no actual data constructors. Unsurprisingly, we will see that non-normalising terms such as $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$ can only be typed with $\bullet^\infty$. Without Condition 4, Ω could be given any type.

We adopt the usual conventions regarding arrow types (which associate to the right) and assume the following precedence among constructors: $\to$, $\multimap$, $\times$, $\mathtt{IO}$, $\bullet$ with $\mathtt{IO}$ and $\bullet$ having the highest precedence. We also need a notion of duality to relate the session types associated with peer endpoints. Our definition extends the one of [12] in the obvious way to delayed types. More precisely, the *dual* of a session type T is the session type $\overline{T}$ coinductively defined by the equations:

$$\overline{\mathtt{end}} = \mathtt{end} \qquad \overline{\mathtt{?}t.T} = \mathtt{!}t.\overline{T} \qquad \overline{\mathtt{!}t.T} = \mathtt{?}t.\overline{T} \qquad \overline{\bullet T} = \bullet\overline{T}$$

Sometimes we will write $\bullet^n t$ in place of $\underbrace{\bullet\cdots\bullet}_{n\text{-times}} t$.

3.2 Typing Rules

We show the typing of expressions and processes. First we assign types to constants:

$$\begin{array}{llll}
\mathtt{unit} & : \mathtt{Unit} & \mathtt{send} & : \mathtt{!}t.T \to t \multimap \mathtt{IO}\ T \\
\mathtt{return} & : t \to \mathtt{IO}\ t & \mathtt{recv} & : \mathtt{?}t.T \to \mathtt{IO}\ (t \times T) \\
\mathtt{open} & : \langle T \rangle \to \mathtt{IO}\ T & \mathtt{future} & : \bullet^n(\mathtt{IO}\ t) \to \mathtt{IO}\ \bullet^n t
\end{array}$$

$$\begin{array}{lll}
\mathtt{pair} : t \to s \multimap t \times s & & \text{if } \mathsf{lin}(t) \\
\mathtt{pair} : t \to s \to t \times s & & \text{if } \mathsf{un}(t) \\
\mathtt{bind} : \mathtt{IO}\ t \to (t \multimap \mathtt{IO}\ s) \multimap \mathtt{IO}\ s & &
\end{array}$$

Each constant $\mathtt{k} \neq \mathtt{unit}$ is polymorphic and we use $\mathsf{types}(\mathtt{k})$ to denote the set of types assigned to $\mathtt{k}$, e.g. $\mathsf{types}(\mathtt{return}) = \cup_t \{t \to \mathtt{IO}\ t\}$.

Table 4. Typing rules for expressions.

$$[\bullet\mathrm{I}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : \bullet t} \qquad [\mathrm{CONST}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathtt{k} : t}\ t \in \mathsf{types}(\mathtt{k}) \qquad [\mathrm{AXIOM}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma, u : t \vdash u : t}$$

$$[\to\mathrm{I}]\ \frac{\Gamma, x : \bullet^n t \vdash e : \bullet^n s}{\Gamma \vdash \lambda x.e : \bullet^n (t \to s)}\ \mathsf{un}(\Gamma) \qquad [\to\mathrm{E}]\ \frac{\Gamma_1 \vdash e_1 : \bullet^n (t \to s) \qquad \Gamma_2 \vdash e_2 : \bullet^n t}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : \bullet^n s} \qquad [\multimap\mathrm{I}]\ \frac{\Gamma, x : \bullet^n t \vdash e : \bullet^n s}{\Gamma \vdash \lambda x.e : \bullet^n (t \multimap s)}$$

$$[\multimap\mathrm{E}]\ \frac{\Gamma_1 \vdash e_1 : \bullet^n (t \multimap s) \qquad \Gamma_2 \vdash e_2 : \bullet^n t}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : \bullet^n s} \qquad [\times\mathrm{E}]\ \frac{\Gamma_1 \vdash e : \bullet^n (t_1 \times t_2) \qquad \Gamma_2, x : \bullet^n t_1, y : \bullet^n t_2 \vdash f : \bullet^n s}{\Gamma_1 + \Gamma_2 \vdash \mathtt{split}\ e\ \mathtt{as}\ x, y\ \mathtt{in}\ f : \bullet^n s}$$

The types of `unit` and `return` are as expected. The type schema of `bind` is similar to the type it has in Haskell, except for the two linear arrows. The leftmost linear arrow allows linear functions as the second argument of `bind`. The rightmost linear arrow is needed to satisfy Condition 1 of Definition 1, being `IO` t linear. The type of `pair` is also familiar, except that the second arrow is linear or unlimited depending on the first element of the pair. If the first element of the pair is a linear expression, then it can (and actually must) be used for creating exactly one pair. The types of `send` and `recv` are almost the same as in [11], except that these primitives return I/O actions instead of performing them as side effects. The type of `open` is standard and obviously justified by its operational semantics. The most interesting type is that of `future`, which commutes delays and the `IO` type constructor. Intuitively, `future` applied to a delayed I/O action returns an immediate I/O that yields a delayed expression. This fits with the semantics of `future`, since its argument is evaluated in a separate thread and the one invoking `future` can proceed immediately with a placeholder for the delayed expression. If the body of the new thread reduces to `return` e, then e substitutes the placeholder.

The typing judgements for expressions have the shape $\Gamma \vdash e : t$, where *typing environments* (for used resources) Γ are mappings from variables to types, from shared channels to shared channel types, and from endpoints to session types:

$$\Gamma \quad ::= \quad \emptyset \quad | \quad \Gamma, x : t \quad | \quad \Gamma, a : \langle T \rangle \quad | \quad \Gamma, a^p : T$$

A typing environment Γ is *linear*, notation $\mathsf{lin}(\Gamma)$, if there is $u : t \in \Gamma$ such that $\mathsf{lin}(t)$; otherwise Γ is *unlimited*, notation $\mathsf{un}(\Gamma)$. As in [11], we use a (partial) combination operator $+$ for environments, that prevents names with linear types from being duplicated. Formally the environment $\Gamma + \Gamma'$ is defined inductively on Γ' by

$$\begin{array}{c} \Gamma + \emptyset = \Gamma \\ \Gamma + (\Gamma', u : t) = (\Gamma + \Gamma') + u : t \end{array} \quad \text{where} \quad \Gamma + u : t = \begin{cases} \Gamma, u : t & \text{if } u \notin \mathsf{dom}(\Gamma), \\ \Gamma & \text{if } u : t \in \Gamma \text{ and } \mathsf{un}(t), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The typing axioms and rules for expressions are given in Table 4. They are essentially the same as those found in [11], except for two crucial details. First of all, each

rule allows for an arbitrary delay in front of the types of the entities involved. Intuitively, the number of $\bullet$'s represents the delay at which a value becomes available. So for example, rule [→I] says that a function which accepts an argument x of type t delayed by n and produces a result of type s delayed by the same n has type $\bullet^n(t \to s)$, that is a function delayed by n that maps elements of t into elements of s. The second difference with respect to the type system in [11] is the presence of rule [•I], which allows to further delay a value of type t. Crucially, it is not possible to *anticipate* a delayed value: if it is known that a value will only be available with delay n, then it will also be available with any delay $m \geq n$, but not earlier. Using rule [•I], we can derive that the fixed point combinator $\mathtt{fix} = \lambda y.(\lambda x.y\ (x\ x))(\lambda x.y\ (x\ x))$ has type $(\bullet t \to t) \to t$, by assigning to the variable x the type s such that $s = \bullet s \to t$ [19]. The side condition $\mathsf{un}(\Gamma)$ in [CONST], [AXIOM], and [→I] is standard [11].

It is possible to derive the following types for the functions in Section 1:

$$\mathtt{from} : \mathtt{Nat} \to \mathtt{S}_{\mathtt{Nat}} \qquad \mathtt{stream} : \mathtt{Out}_{\mathtt{Nat}} \to \mathtt{S}_{\mathtt{Nat}} \to \mathtt{IO}\ \bullet^{\infty} \qquad \mathtt{display} : \mathtt{In}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}$$

where, in the derivation for display, we assume type $\mathtt{S}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}$ for g. We show the most interesting parts of this derivation. We use the following rules, which are easily derived from those in Table 4 and the types of the constants.

$$[\text{FIX}]\ \frac{\Gamma, x : \bullet t \vdash e : t}{\Gamma \vdash \mathtt{fix}\ \lambda x.e : t}\ \mathsf{un}(\Gamma) \qquad [\text{BIND}]\ \frac{\Gamma_1 \vdash e_1 : \bullet^n(\mathtt{IO}\ t) \qquad \Gamma_2 \vdash e_2 : \bullet^n(t \multimap \mathtt{IO}\ s)}{\Gamma_1 + \Gamma_2 \vdash e_1 \gg\!= e_2 : \bullet^n \mathtt{IO}\ s}$$

$$[\text{FUTURE}]\ \frac{\Gamma \vdash e : \bullet^{n+m} \mathtt{IO}\ t}{\Gamma \vdash \mathtt{future}\ e : \bullet^n \mathtt{IO}\ \bullet^m t} \qquad [\times \to \text{I}]\ \frac{\Gamma, x_1 : \bullet^n t_1, x_2 : \bullet^n t_2 \vdash e : \bullet^n s}{\Gamma \vdash \lambda\langle x_1, x_2\rangle.e : \bullet^n(t_1 \times t_2 \to s)}\ \mathsf{un}(\Gamma)$$

In order to derive the type of display we desugar its recursive definition in Section 1 as $\mathtt{display} = \mathtt{fix}\ (\lambda x.\lambda y.e)$, where

$$e = e_1 \gg\!= e_2 \qquad \begin{array}{l} e_1 = \mathtt{recv}\ y \\ e_2 = \lambda\langle z, y'\rangle.e_3 \gg\!= e_4 \end{array} \qquad \begin{array}{l} e_3 = \mathtt{future}(x\ y') \\ e_4 = \lambda z s.g\langle z, z s\rangle \end{array}$$

Now we derive

$$[\text{FIX}]\ \frac{[\text{BIND}]\ \dfrac{\dfrac{\vdots}{\Gamma_1 \vdash e_1 : \mathtt{IO}\ (\mathtt{Nat} \times \bullet \mathtt{In}_{\mathtt{Nat}})} \qquad [\times \to \text{I}]\ \dfrac{\dfrac{\nabla}{\Gamma, \Gamma_2, \Gamma_3 \vdash e_3 \gg\!= e_4 : \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}}}{\Gamma \vdash e_2 : (\mathtt{Nat} \times \bullet \mathtt{In}_{\mathtt{Nat}}) \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}}}{[\rightarrow\text{I}]\ \dfrac{\Gamma, y : \mathtt{In}_{\mathtt{Nat}} \vdash e : \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}}{\Gamma \vdash \lambda y.e : \mathtt{In}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}}}}{\vdash \mathtt{display} : \mathtt{In}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}}$$

where $\Gamma = x : \bullet(\mathtt{In}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}})$, $\Gamma_1 = y : \mathtt{In}_{\mathtt{Nat}}$, $\Gamma_2 = y' : \bullet \mathtt{In}_{\mathtt{Nat}}$ and $\Gamma_3 = z : \mathtt{Nat}, g : \mathtt{S}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}$. The derivation ∇ is as follows.

$$[\rightarrow\text{E}]\ \frac{\Gamma \vdash x : \bullet(\mathtt{In}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}) \qquad \Gamma_2 \vdash y' : \bullet \mathtt{In}_{\mathtt{Nat}}}{[\text{FUTURE}]\ \dfrac{\Gamma, \Gamma_2 \vdash x\ y' : \bullet \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}}{\Gamma, \Gamma_2 \vdash e_3 : \mathtt{IO}\ \bullet \mathtt{S}_{\mathtt{Nat}}}} \qquad \frac{\vdots}{\Gamma_3 \vdash e_4 : \bullet \mathtt{S}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}}$$
$$[\text{BIND}]\ \frac{}{\Gamma, \Gamma_2, \Gamma_3 \vdash e_3 \gg\!= e_4 : \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}}$$

Table 5. Typing rules for processes.

$$\text{[THREAD]}\ \frac{\Gamma \vdash e : \bullet^n(\mathtt{IO}\ t)}{\Gamma \vdash x \Leftarrow e \triangleright x : \bullet^n t}\ x \notin \mathsf{dom}(\Gamma) \qquad \text{[SERVER]}\ \frac{\Gamma \vdash e : \overline{T} \to \mathtt{IO}\ t}{\Gamma + a : \langle T \rangle \vdash \mathtt{server}\ a\ e \triangleright a : \langle T \rangle}\ \begin{array}{l}\mathsf{shared}(\Gamma)\\ \mathsf{un}(t)\end{array}$$

$$\text{[PAR]}\ \frac{\Gamma_1 \vdash P_1 \triangleright \Delta_1 \qquad \Gamma_2 \vdash P_2 \triangleright \Delta_2}{\Gamma_1 + \Gamma_2 \vdash P_1 \mid P_2 \triangleright \Delta_1, \Delta_2} \qquad \text{[SESSION]}\ \frac{\Gamma, a^p : T, a^{\overline{p}} : \overline{T} \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)P \triangleright \Delta} \qquad \text{[NEW]}\ \frac{\Gamma, X : t \vdash P \triangleright \Delta, X : t}{\Gamma \vdash (\nu X)P \triangleright \Delta}$$

Note that the types of the premises of [→E] in the above derivation have a • constructor in front. Moreover, `future` has a type that pushes the • inside the `IO`; this is crucial for typing e_4 with $(\bullet\mathtt{S}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}})$. We can assign the type $\bullet\mathtt{S}_{\mathtt{Nat}} \to \mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}}$ to e_4 by guarding the argument z of type $\bullet\mathtt{S}_{\mathtt{Nat}}$ under the constructor `pair`. Without `future`, the expression e_3 >>= e_4 would have type $\bullet(\mathtt{IO}\ \mathtt{S}_{\mathtt{Nat}})$ and `display` would be untypeable.

The typing judgements for processes have the shape $\Gamma \vdash P \triangleright \Delta$, where Γ is a typing environment as before, while Δ is a *resource environment*, keeping track of the resources defined in P. In particular, Δ maps the names of threads and servers in P to their types and it is defined by

$$\Delta \quad ::= \quad \emptyset \quad \mid \quad \Delta, x : t \quad \mid \quad \Delta, a : \langle T \rangle$$

Table 5 gives the typing rules for processes. A thread is well typed if so is its body, which must be an I/O action. The type of a thread is that of the result of its body, where the delay moves from the I/O action to the result. The side condition makes sure that the thread is unable to use the very value that it is supposed to produce. The resulting environment for defined resources associates the name of the thread with the type of the action of its body. A server is well typed if so is its body e, which must be a function from the dual of T to an I/O action. This agrees with the reduction rule of the server, where the application of e to an endpoint becomes the body of a new thread each time the server is invoked. It is natural to forbid occurrences of free variables and shared channels in server bodies. This is assured by the condition $\mathsf{shared}(\Gamma)$, which requires Γ to contain only shared channels. Clearly $\mathsf{shared}(\Gamma)$ implies $\mathsf{un}(\Gamma)$, and then we can type the body e with a non linear arrow. The type of the new thread (which will be t if e has type $\overline{T} \to \mathtt{IO}\ t$) must be unlimited, since a server can be invoked an arbitrary number of times. The environment $\Gamma + a : \langle T \rangle$ in the conclusion of the rule makes sure that the type of the server as seen by its clients is consistent with its definition.

The remaining rules are conventional. In a parallel composition we require that the sets of entities (threads and servers) defined by P_1 and P_2 are disjoint. This is enforced by the fact that the respective resource environments Δ_1 and Δ_2 are combined using the operator $\_,\_$ which (as usual) implicitly requires that $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) = \emptyset$. The restriction of a session channel a introduces associations for both its endpoints a^+ and a^- in the typing environment with dual session types, as usual. Finally, the restriction of a bindable name X introduces associations in both the typing and the resource environment with the same type t. This makes sure that in P there is exactly

one definition for X, which can be either a variable which names a thread or a shared channel which names a server, and that every usage of X is consistent with its definition.

4 Main Results

In this section we state the main properties enjoyed by typed SID programs. The first expected property is that reduction of expressions preserves their types.

Theorem 1 (Subject Reduction for Expressions). *If* $\Gamma \vdash e : t$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : t$.

Besides the usual substitution lemma, the proof of the above theorem needs the delay lemma, which states that if an expression e has type t from Γ, then it has type $\bullet t$ from $\bullet\Gamma$. This property reflects the fact that we can only move forward in time.

As informally motivated in Section 3, the type constructor $\bullet$ controls recursion and guarantees normalisation of any expression that has a type different from $\bullet^{\infty}$.

Theorem 2 (Normalisation of Typeable Expressions). *If* $\Gamma \vdash e : t$ *and* $t \neq \bullet^{\infty}$, *then* e *reduces (in zero or more steps) to a normal form.*

The proof of Theorem 2 makes use of a type interpretation indexed on the set of natural numbers, similar to the one given in [19]. Note that, since SID is lazy, expressions such as `return` e and $\langle e\,,f\rangle$ are in normal form for all e and f.

An *initial* process models the beginning of a computation and it is formally defined as a closed, well-typed process P such that

$$P \equiv (\nu x a_1 \cdots a_m)(x \Leftarrow e \mid \texttt{server}\ a_1\ e_1 \mid \cdots \mid \texttt{server}\ a_m\ e_m)$$

By definition, an initial process does not contain undefined names (hence it is typeable from the empty environment) and consists of only one thread x – usually called "main" in most programming languages – and an arbitrary number of servers. In particular, typeability guarantees that all bodies normalise and all `open`'s refer to existing servers.

We say that a process is *reachable* if it is the reduct of an initial process. Unlike an initial process, a reachable process may have several threads running in parallel, resulting from either service invocation or `future`.

Theorem 3 (Subject Reduction for Processes). *All reachable processes are typeable.*

The most original and critical aspect of the proof is to check that reachable processes do not have circular dependencies on session channels and variables. The absence of circularities can be properly formalized by means of a judgement that characterises the sharing of names among threads, inspired by the typing of the parallel composition given in [16]. Intuitively, it captures the following properties of reachable processes and makes them suitable for proving both subject reduction and progress:

1. two threads can share at most one session channel;
2. distinct endpoints of a session channel always occur in different threads;
3. if the name of one thread occurs in the body of another thread, then these threads cannot share session channels nor can the first thread mention the second.

Next, we show several examples of processes that are irrelevant to us because, in spite of being typeable, they are not reachable. Examples (4.1) and (4.2) violate condition (3), (4.3) violates condition (1), and (4.4) violates condition (2).

The first example is given by the process

$$(\nu xy)(x \Leftarrow \texttt{return}\ y \mid y \Leftarrow \texttt{return}\ x) \tag{4.1}$$

which is well typed by assigning both x and y any unlimited type, whereas $(\nu x)(x \Leftarrow \texttt{return}\ x)$, which is its reduct, is ill typed, because the thread name x occurs free in its body (*cf.* the side condition of [THREAD]). Another paradigmatic example is

$$x \Leftarrow \texttt{send}\ a^{+} y \mid y \Leftarrow \texttt{recv}\ a^{-} \tag{4.2}$$

which is well typed in the environment $a^{+} : !t.\texttt{end}, a^{-} : ?t.\texttt{end}, y : t$, where $t = \bullet(t \times \texttt{end})$, and which reduces to $x \Leftarrow \texttt{return}\ a^{+} \mid y \Leftarrow \texttt{return}\ \langle y, a^{-} \rangle$. Again, the reduct is ill typed because the name y of the thread occurs free in its body.

Another source of problems that usually requires specific handling [2,4] is that there exist well-typed processes that are (or reduce to) configurations where mutual dependencies between sessions and/or thread names prevent progress. For instance, both

$$(\nu xyab)(x \Leftarrow \texttt{send}\ a^{+}\ 4 \gg\!= \lambda x.\texttt{recv}\ b^{-} \mid y \Leftarrow \texttt{send}\ b^{+}\ 2 \gg\!= \lambda x.\texttt{recv}\ a^{-}) \tag{4.3}$$

$$(\nu xa)(x \Leftarrow \texttt{recv}\ a^{-} \gg\!= \lambda \langle y, z \rangle.\texttt{send}\ a^{+}\ y) \tag{4.4}$$

are well typed but also deadlocked. Again, processes like this one are not reachable hence they are not a concern in our case.

We now turn our attention to the progress property. A computation stops when there are no threads left. Recall that the reduction rule [R-RETURN] (*cf.* Table 2) erases threads. Since servers are permanent we say that a process P is *final* if

$$P \equiv (\nu a_1 \ldots a_m)(\texttt{server}\ a_1\ e_1 \mid \ldots \mid \texttt{server}\ a_m\ e_m)$$

In particular, the idle process is final, since m can be 0.

We can state the progress property as follows:

Theorem 4 (Progress of Reachable Processes). *A reachable process either reduces or it is final. Moreover a non-terminating reachable process reduces in a finite number of steps to a process to which one of the rules* [R-OPEN], [R-COMM] *or* [R-FUTURE] *can be applied.*

In other words, every infinite reduction of a reachable process performs infinitely many communications and/or spawns infinitely many threads. The proof of Theorem 4 requires to define a precedence between threads and prove that this relation is acyclic.

As an example, let

$$Q \;=\; (\nu prod\, cons\, a\, c)(P \mid \texttt{server}\ a\ \ \lambda y.\texttt{display}\ y)$$

where

$$P \;=\; prod \Leftarrow \texttt{stream}\ c^{+}\ (\texttt{from}\ 0) \mid cons \Leftarrow \texttt{display}\ c^{-}$$

is the process discussed in the Introduction. It is easy to verify that

$$P_0 \;=\; (\nu\, prod\, a)(prod \Leftarrow \texttt{open}\ a \gg\!= \lambda y.\texttt{stream}\ y\ (\texttt{from}\ 0) \mid \texttt{server}\ a\ \ \lambda y.\texttt{display}\ y)$$

reduces to process Q. Note that P_0 is typeable, and indeed an initial process. Hence, by Theorems 3 and 4, process Q is typeable and has progress.

The last property of SID we discuss is the diamond property [24, §30.3].

Theorem 5 (Confluence of Reachable Processes). *Let P be a reachable process. If $P \longrightarrow P_1$ and $P \longrightarrow P_2$, then there is P_3 such that $P_1 \longrightarrow P_3$ and $P_2 \longrightarrow P_3$.*

The proof is trivial for expressions, since there is only one redex at each reduction step. However, for processes we may have several redexes to contract at a time and the proof requires to analyse these possibilities. The fact that we can mix pure evaluations and communications and still preserve determinism is of practical interest.

We conclude this section discussing two initial processes whose progress is somewhat degenerate. The first one realises an infinite sequence of *delegations* (the act of sending an endpoint as a message), thereby postponing the use of the endpoint forever:

$$\begin{aligned}\texttt{badserver} \stackrel{\text{def}}{=} (\nu xab)(x \Leftarrow \texttt{open}\ a \gg\!= \texttt{loop1} \mid \\ \texttt{server}\ \ a\ \lambda y.\texttt{open}\ b \gg\!= \texttt{loop2}\ y \mid \texttt{server}\ \ b\ \texttt{recv})\end{aligned}$$

where

$$\begin{aligned}\texttt{loop1} \stackrel{\text{def}}{=} \texttt{fix}\ \lambda f.\lambda x.\texttt{recv}\ x \gg\!= \lambda y.\texttt{split}\ y\ \texttt{as}\ y_1, y_2\ \texttt{in}\ \texttt{send}\ y_2\ y_1 \gg\!= \\ \lambda z.\texttt{future}\ (f z)\end{aligned}$$

$$\begin{aligned}\texttt{loop2} \stackrel{\text{def}}{=} \texttt{fix}\ \lambda g.\lambda y x.\texttt{send}\ x\ y \gg\!= \lambda z.\texttt{recv}\ z \gg\!= \\ \lambda u.\texttt{split}\ u\ \texttt{as}\ u_1, u_2\ \texttt{in}\ \texttt{future}\ (g u_1 u_2)\end{aligned}$$

We have that $\texttt{loop1} : \mathsf{RS}_t \to \texttt{IO}\ \bullet^\infty$ and $\texttt{loop2} : t \to \mathsf{SR}_t \multimap \texttt{IO}\ \bullet^\infty$ where $\mathsf{RS}_t = ?t.!t.\bullet\ \mathsf{RS}_t$ and $\mathsf{SR}_t = !t.?t.\bullet\ \mathsf{SR}_t$. Since no communication ever takes place on the session created with server b, `badserver` violates the progress property as defined in [8].

The second example is the initial process $(\nu x)(x \Leftarrow \Omega_{\texttt{future}})$, where $\Omega_{\texttt{future}} = \texttt{fix future}$. This process only creates new threads.

5 Conclusions

This work addresses the problem of studying the interaction between communications and infinite data structures by means of a calculus that combines sessions with lazy evaluation. A distinguished feature of SID is the possibility of modelling computations in which infinite communications interleave with the production and consumption of infinite data (*cf.* the examples in Section 1). Our examples considered infinite streams for simplicity. However, more general infinite data structures can be handled in SID. An evaluation of the expressiveness of SID in dealing with (distributed) algorithms based on such structures is scope for future investigations.

The typing discipline we have developed for SID guarantees normalisation of expressions with a type other than $\bullet^\infty$ and progress of (reachable) processes, besides the standard properties of sessions (communication safety, protocol fidelity, determinism). The type system crucially relies on a modal operator $\bullet$ which has been used in a number of previous works [19,14,26,3] to ensure productivity of well-typed expressions. In this

paper, we have uncovered for the first time some intriguing interactions between this operator and the typing of impure expressions with the monadic `IO` type constructor. Conventionally, the type of `future` primitive is simply `IO` $t \to$ `IO` t and says nothing about the semantics of the primitive itself. In our type system, the type of `future` reveals its effect as an operator that turns a delayed computation into another that can be performed immediately, but which produces a delayed result.

As observed at the end of Section 1 and formalised in Theorem 4, our notion of progress sits somehow in between deadlock and lock freedom. It would be desirable to strengthen the type system so as to guarantee the (eventual) execution of all pending communications and exclude, for instance, the degenerate examples discussed at the end of Section 4. This is relatively easy to achieve in conventional process calculi, where expressions only consist of names or ground values [2,4,20], but it is far more challenging in the case of SID, where expressions embed the λ-calculus. We conjecture that one critical condition to be imposed is to forbid postponing linear computations, namely restricting the application of $[\bullet\mathrm{I}]$ to non-linear types. Investigations in this direction are left for future work.

Another obvious development, which is key to the practical applicability of our theory, is the definition of a type inference algorithm for our type system. In this respect, the modal operator $\bullet$ is challenging to deal with because it is intrinsically non-structural, not corresponding to any expression form in the calculus.

Acknowledgments. We are grateful to the anonymous reviewers for their useful suggestions, which led to substantial improvements.

References

1. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The Call-by-Need Lambda Calculus. In R. K. Cytron and P. Lee, editors, *proceedings of POPL'95*, pages 233–246. ACM Press, 1995.
2. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In F. van Breugel and M. Chechik, editors, *proceedings of CONCUR'08*, LNCS 5201, pages 418–433. Springer, 2008.
3. A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair Reactive Programming. In S. Jagannathan and P. Sewell, editors, *proceedings of POPL'14*, pages 361–372. ACM Press, 2014.
4. M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.
5. T. Coquand. Infinite Objects in Type Theory. In H. Barendregt and T. Nipkow, editors, *proceedings of TYPES'93*, LNCS 806, pages 62–78. Springer, 1993.
6. B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.
7. L. Cruz-Filipe, I. Lanese, F. Martins, A. Ravara, and V. Vasconcelos. The Stream-based Service-centred Calculus: a Foundation for Service-oriented Programming. *Formal Aspects of Computing*, 26(12):865–918, 2014.
8. P.-M. Deniélou and N. Yoshida. Dynamic Multirole Session Types. In T. Ball and M. Sagiv, editors, *proceedings of POPL'11*, pages 435–446. ACM Press, 2011.

9. M. Dezani-Ciancaglini and U. de' Liguoro. Sessions and Session Types: an Overview. In C. Laneve and J. Su, editors, *proceedings of WS-FM'09*, LNCS 6194, pages 1–28. Springer, 2009.
10. C. Flanagan and M. Felleisen. The Semantics of Future and an Application. *Journal of Functional Programming*, 9(1):1–31, 1999.
11. S. J. Gay and V. T. Vasconcelos. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming*, 20(1):19–50, 2010.
12. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In C. Hankin, editor, *proceedings of ESOP'98*, LNCS 1381, pages 122–138. Springer, 1998.
13. J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
14. N. Krishnaswami and N. Benton. Ultrametric Semantics of Reactive Programs. In M. Grohe, editor, *proceedings of LICS'11*, pages 257–266. IEEE, 2011.
15. N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order functional reactive programming in bounded space. In *proceedings of POPL'12*, pages 45–58. ACM Press, 2012.
16. S. Lindley and J. G. Morris. A Semantics for Propositions as Sessions. In J. Vitek, editor, *proceedings of ESOP'15*, LNCS 9032, pages 560–584. Springer, 2015.
17. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 2012.
18. J. Maraist, M. Odersky, and P. Wadler. The Call-by-Need Lambda Calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.
19. H. Nakano. A Modality for Recursion. In M. Abadi, editor, *proceedings of LICS'00*, pages 255–266. IEEE, 2000.
20. L. Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In T. A. Henzinger and D. Miller, editors, *proceedings of LICS'14*, pages 72:1–72:10. ACM Press, 2014.
21. S. Peyton Jones. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell. In T. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.
22. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In H. Boehm and G. L. Steele Jr., editors, *proceedings of POPL'96*, pages 295–308. ACM Press, 1996.
23. S. Peyton Jones and P. Wadler. Imperative Functional Programming. In M. S. V. Deusen and B. Lang, editors, *proceedings of POPL'93*, pages 71–84. ACM Press, 1993.
24. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
25. D. Sabel and M. Schmidt-Schauß. A Contextual Semantics for Concurrent Haskell with Futures. In P. Schneider-Kamp and M. Hanus, editors, *proceedings of PPDP'11*, pages 101–112. ACM Press, 2011.
26. P. Severi and F.-J. de Vries. Pure Type Systems with Corecursion on Streams: from Finite to Infinitary Normalisation. In P. Thiemann and R. B. Findler, editors, *proceedings of ICFP'12*, pages 141–152. ACM Press, 2012.
27. P. Severi, L. Padovani, E. Tuosto, and M. Dezani-Ciancaglini. On sessions and infinite data. Technical report, Universiy of Leicester and Università di Torino, 2016. Available at `https://hal.archives-ouvertes.fr/hal-01297293`.
28. B. Toninho, L. Caires, and F. Pfenning. Corecursion and Non-divergence in Session-Typed Processes. In M. Maffei and E. Tuosto, editors, *proceedings of TGC'14*, LNCS 8902, pages 159–175. Springer, 2014.
29. V. T. Vasconcelos. Fundamentals of Session Types. *Information and Computation*, 217:52–70, 2012.
30. C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.